

SHARPE: Variation-Aware Formal Statistical Timing Analysis in RTL

Abstract—Variations in timing can occur due to multiple sources on a chip. Many circuit level statistical techniques are used to analyze timing in the presence of these sources of variation. At the system (higher) level of design, however, timing estimation/verification is not performed. The design at the Register Transfer Level (RTL) is unaware of the underlying statistics and timing variations. It is desirable to have “variation awareness” at the higher level, and estimate block level delay distributions early in the design cycle, to evaluate design choices quickly and minimize post-synthesis simulation costs.

In this paper, we introduce SHARPE, a rigorous, systematic timing analysis/verification methodology and tool flow to find statistical delay invariants in RTL. We treat the RTL source code as a program and use static program analysis techniques to compute probabilities. We model the probabilistic RTL modules as Discrete Time Markov Chains (DTMCs) that are then checked formally for probabilistic invariants using PRISM, a probabilistic model checker. Our technique is illustrated on the RTL description of the datapath of OR1200, an open source embedded processor.

I. INTRODUCTION

Adaptive techniques like voltage and frequency scaling, process variations due to shrinking chip geometries and input variations for accommodating average-case timing design contribute significantly to the stochastic nature of contemporary chips [1]. At present, this stochastic nature is addressed only at the lower levels of the system design cycle, where appropriate circuit design measures are taken to allow for variation-dependence in timing. Timing verification methods include static timing analysis and statistical static timing analysis [2] [3] [4] [5]. Some circuit design techniques like [6] [7] use timing analysis to enable better design goals than pessimistic worst case design, and try to tune the circuits to achieve these goals.

Although there is variation-awareness in high level synthesis techniques [8] [9] as well as architecture level power and performance analysis [10], this is not observed in RT-level design verification methodologies. The behavioral levels of the design as in Register Transfer Level (RTL) lack awareness of the underlying stochastic nature of the circuit. Although there are multiple techniques for checking functional specifications, to the best of our knowledge, there are no well-known methods to verify the adherence of a design to a given timing specification in RTL.

In order to express functional correctness without timing, a deterministic model is sufficient to express desired properties or invariants across the design. However, this purely deterministic model of functionality at the higher level, does not

address the multiple growing sources of variation at the lower levels of design.

In order to develop an alternate notion of design correctness, the statistical nature of delay variations needs to be incorporated. It is therefore necessary to establish *probabilistic invariants* [11] [12] across the design. These invariants would attest certain timing properties of the design that hold true in the presence of the underlying variations. They can answer questions of the type: “What is the probability of a block/module signal meeting a timing specification?” or “What is the average delay of a signal in a block/module?”.

This information, if available in RTL, can provide quick and early estimates of delay distribution of different blocks, thereby facilitating better design choices and reducing the overhead in post-synthesis simulation methods [13]. Since delay is an artifact of gate level elements, we obtain delay macromodels [14] [13] from the gate level. Typically, these macromodels provide estimates in RTL that are within 20% of the actual measurements obtained at gate level.

We introduce SHARPE (Statistical High Level Analysis and Rigorous Performance Estimation). SHARPE is a CAD tool flow and methodology for computing probabilistic delay distribution formally in RTL. The steps in SHARPE are as follows. We determine signal correlations using static analysis of RTL source code. We then convert an RTL module into a Discrete Time Markov Chain Model (DTMC) [15] with the transition probabilities derived from input probability distributions as well as the static analysis. This DTMC is converted to a “reward model” where the states are annotated by a cost function, which in our case is delay/timing. We obtain delay macromodels from the gate level that we can use to assign rewards to the RTL DTMC model. We use a probabilistic model checking engine, PRISM [16] to compute probabilistic invariants with respect to timing.

Markov chains have frequently been used to compute high level system performance and power [17] [18]. They have also been used to design circuits with high error tolerance [19]. However, in our context, DTMCs are being used to rigorously generate delay related invariants.

SHARPE can be thought of as a formal statistical static timing analysis in RTL. Although conceptually, the technique can consider any source of statistical behavior, in this work, we consider input variations as the primary source of statistics. In the context of “better-than-worst-case-design”, SHARPE addresses an important CAD challenge [20].

An important part of our methodology is the formal verification of probabilistic timing. When posed with a query the

formal engine explores all possible transitions of the RTL-DTMC model. This makes our analysis high in confidence as opposed to simulation based methods for generating probability distributions.

Some niche circuit design techniques [21] rely on the information from the underlying statistical nature. These techniques inject errors into the design, and view the error tolerance of their designs at the system level. Simulating the design at the system level repeatedly is time consuming and inaccurate. These techniques can benefit immensely from our SHARPE analysis, since the effect of injecting errors can be analyzed at a system level exhaustively and very quickly.

The main contributions of this work are as follows.

- We introduce with SHARPE, the notion of “variation-aware” design verification in RTL. We model the RTL as a DTMC and compute statistical timing invariants for block/module outputs. Our technique can be thought of as a statistical static timing analysis done with behavioral level models.
- In this work, we propose a CAD flow for verification of “better-than-worst-case” design paradigms, since we consider input variations as the source of stochastic behavior. Our technique can be used to model different sources of variation as well.
- We present a static analysis technique on the RTL source code to automatically compute inter-signal dependencies and signal correlations for creating a DTMC model.
- We present a delay macromodeling methodology that captures the delay distribution of an RTL operator as a function of input variations.
- We use probabilistic model checking to compute the statistical timing invariants, thereby making the delay distribution analysis rigorous and high-confidence, as opposed to non-analytical simulation based techniques.
- SHARPE enables designers to get early and quick estimates of statistical timing analysis, without going through gate level synthesis and post-synthesis analyses. We show that the estimates from SHARPE are well within 20% of the delay distribution curves extracted from gate-level simulations.

II. BACKGROUND CONCEPTS

An n -bit variable v can be assigned any one of 2^n possible values with associated probabilities. Collectively, these define the *probability distribution* (called PMF) of v . The *expected* value of v is the probability-weighted sum of the possible values. The *joint probability* of a set of variables is the probability with which a set of values are collectively assigned to the variables. Two variables a and b are *independent* if the probability of an assignment to a is not affected by the assignment to b . Variables that are not independent are called *correlated variables*. Probability distributions that do not vary across time are said to be *stationary*.

We assume knowledge of the distribution of primary input variables and that they are independently distributed. We also

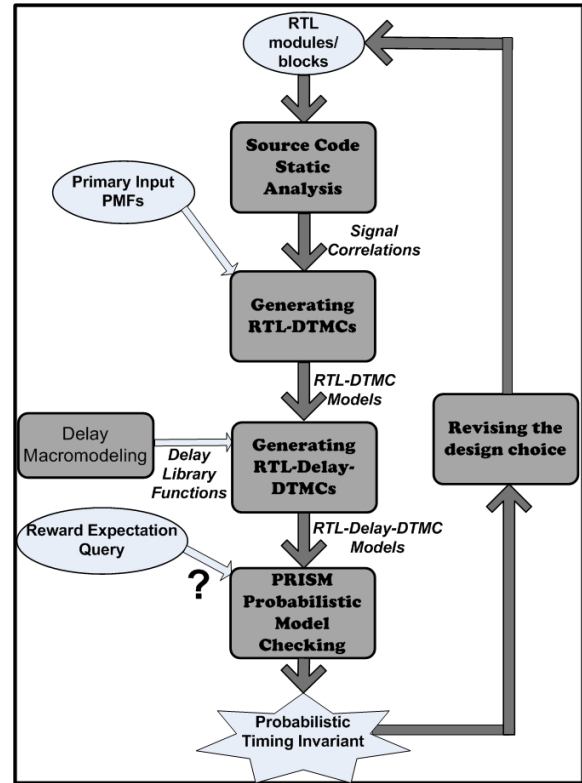


Fig. 1. The SHARPE tool flow.

assume stationary¹ probability distributions for our inputs, and therefore for all variables in the system.

In order to find the PMFs of a set of variables O from the PMFs of a set of variables I , we need to find the function f such that $O = f(I)$. We also need the joint probability distributions of the variables in I .

Definition 1: If the variables in I are independent, their joint PMF is simply a product of their individual PMFs.

III. SHARPE: METHODOLOGY AND TOOL FLOW

In this section, we describe each component of the SHARPE toolflow (Figure 1) in detail.

A. Delay Modeling

We consider RTL designs written in Verilog HDL. We view the RTL design as a Verilog program² [22] on which we can perform static analysis techniques.

To restrict the timing analysis to RTL, we define delay in terms of the RTL statements of interest. We consider the synthesizable subset of Verilog for our analysis. A Verilog program statement is a conditional (*if-else*, *case*) statement or an assignment.

We consider RTL blocks/modules as the basic units for delay computation. A module usually performs one or more independent functions and has inputs and outputs defined.

¹A function of stationary variables is also stationary [15].

²In this paper, we use RTL design interchangeably with Verilog program; similarly we use signal and variable interchangeably.

Every statement in a module uses RTL operators (addition, subtraction, bitwise operators etc). If a variable is on the right hand side of an assignment to a variable v , it is an *operand*. The set of all the operands is called $RHS(v)$.

In order to find delay of a Verilog module (RTL block), we need a delay characterization of all the statements that appear in the module. The delay of a statement depends on the operation that it performs and the values of the operands (elements of set $RHS(v)$). For every RTL operator, we determine a function that computes the delay as a function of the values of its operands. We refer to this function as a *delay macromodel* for that operator.

1) *Macromodels*: In order to obtain delay macromodels, we first synthesize each RTL operator into logic gates and find delay of this implementation for various input transition patterns by gate-level simulation.

We tabulate the delays per input pattern of this implementation. In order to abstract this tabular information into a higher level, we need to find a function that estimates the delay of the operator as a function of operand transitions. We arrive at polynomial functions that predict the delay of an output with small average error for each RTL operator.

For instance, for a ripple-carry adder implementation of the RTL add operator, we determine a Boolean function to reflect the number of bits that ‘ripple’ through to the MSB. A 5th order polynomial function predicts delay at the MSB as a function of the *ripples*, with very small average error. This is the delay macromodel of the ripple carry implementation of the add operator. We consider only MSB for this function definition since it has most impact. In Section IV, we obtain such functions for other operators as well.

We obtain the delay for multiple gate level implementations for every operator. This macromodeling and delay characterization is a one-time effort and can be done offline for a given technology library. We use the NANGATE 45nm Open Cell Library for obtaining the delays for the gate-level implementations.

Although we present simplistic delay macromodels in this paper, we demonstrate that they provide RTL estimates that are within 20% of the gate level estimates. However, SHARPE is not restricted to these macromodels. Rigorous regression-based techniques [23] can be employed to obtain more complex macromodels that can be used in SHARPE.

2) *Delay of an RTL block*: We model the delay of a Verilog statement that assigns the output to a block. We consider *blocking* as well as *non-blocking* assignments. Blocking assignments assign the value in RHS in the current cycle to the LHS, in the same cycle. Non-blocking assignments postpone this assignment to the next cycle.

Definition 2: The *delay* of an RTL assignment statement is the time taken from the (rising) clock edge for the effect of the statement execution to be observed.

We consider the rising edge of clock as the reference point for measuring delay. In the following code segment, the non-blocking assignment to f is performed with the values of d and e evaluated synchronously at the rising edge of clock.

```
always @(posedge clk)
b <= a;d <= c & b;f <= d + e;
```

Consider the following set of blocking assignments

```
always @(posedge clk)
b = a;d = c & b;f = d + e;
```

The assignment to f is ‘blocked’ till the assignment to d is completed. The AND gate skews the arrival time of operand d with respect to e . In this work, we make an approximation to deal with this situation. If there is a transition at the gate output, we assume that the signal arrival time T_{out} at the output is $T_{out} = T_{op} + \max(T_{in1}, T_{in2})$ where T_{op} is the operator delay and T_{in1}, T_{in2} are the arrival times of the operand signals. The $\max()$ function is a coarse approximation to determine signal arrival time. More sophisticated functions are used at the gate-level and can easily be used with our technique too.

B. Source Code Static Analysis

Once the input signal probability distributions are given, we statically analyze the Verilog program to determine probability distributions for variables of interest *i.e.* module outputs. We analyze the program [22] to identify correlated variables and propagate the probabilistic distributions to other variables in the slice and module outputs.

Let O be the set of output variables and I be the set of input variables in a Verilog program. It is reasonable to expect that the output variables are functions of the input variables.

For an output signal v , the *signal function* $f(v)$ is the symbolic expression that includes inputs, or the ‘formula’ that corresponds to its evaluation.

Since we assume that the input PMFs are independent, the joint PMF of any subset of I that are included in $f(v)$ are easily calculated as in Definition 1.

We statically traverse the source code for computing the signal function for an output variable of interest. For a Verilog statement where a value of v is assigned, $RHS(v)$ gives the set of variables that are included in the right hand side of the assignment statement. These variables are annotated with a value $t - 1$ to denote that we step the design backwards by a step.³

Definition 3: The *support* of a signal function $f(v)$ is the set of all variables in $f(v)$

Definition 4: The support of a signal is independent if all the variables in the support are independent

If the variables in $RHS(v)$ at time t are independent, their joint PMF is calculated as in Definition 1 and the analysis is complete. If the variables in $RHS(v)$ are not independent, we step the circuit backwards by one more step. Now each variable in $RHS(RHS(v))$ is annotated by $t - 2$. If the union over the set of all variables tagged with $t - 2$ is independent, we

³In programs, this ‘stepping back’ refers to moving in the space of the program source code. The hardware interpretation of this is that the ‘previous cycle’ values of variables are being obtained. This is a temporal step back, as opposed to the purely spatial one in software.

define the union to be the support and obtain the corresponding $f(v)$. Since the support is independent, the joint PMF is calculated as in Definition 1 and this can be used to compute the PMF of v . The algorithm terminates when an independent support is obtained, which in the worst case is an independent subset of I .

The static analysis methodology to establish independence of two variables x and y (as in Section II) is as follows. If x and y are inputs, they are independent by assumption. If x and y are not inputs, they are recursively defined as independent if each variable in the union of $RHS(x)$ and $RHS(y)$ is independent.

Conversely, if x and y are correlated, there is at least one common/shared variable among the variables with the same annotation $t - i$.

In order to be able to use probability distributions of variables assigned in one time step over future time steps, we need to assume that these PMFs are stationary, *i.e.* they do not change over time (Section II).

```

module (A,B,C,D,E,F,G,H)
input A,B,C;output D,E,F;
always @(posedge clk)
begin
D <= A ;E <= B & C;F <= D + E;
end
endmodule

```

In the code fragment shown above, let F be the variable whose probability distribution we want to calculate. Here, A, B and C are primary inputs to the system. D and E are the elements of $RHS(F)$. Since we do not know whether D and B are independent, we step back once more and determine that A, B and C are the elements of the union of $RHS(D)$ and $RHS(E)$. Therefore D and E are independent and form the support of F with $f(F) = D + E$.

C. Generating RTL-DTMCs and RTL-Delay-DTMCs

We now describe the process of converting RTL into a finite-state probabilistic system with respect to input variations. We represent the RTL formally as a finite DTMC, which we call an *RTL-DTMC* model. A finite *DTMC* is a finite state machine where each transition is associated with a probability. A *DTMC state* is a unique assignment of values to a set of variables called *state variables*. A transition in a DTMC is a movement from one state to another, *i.e.* an assignment of a different set of values to the state variables. The transition is labeled with the joint probability with which the state variables would be assigned their respective values in the new state.

The RTL-DTMC model when combined with the delay macromodel provides information about the *probabilistic delay distribution* at the module outputs. We call this an *RTL-Delay-DTMC* model.

The probabilistic behaviour of a signal v can be completely represented by the independent support of v , along with its joint PMF. We construct the RTL-DTMC with the support being the state variables. Since the support is independent, we

obtain the transition probabilities by taking the product of the individual probabilities. In every state, the value assigned to v is given by $f(v)$, which is a symbolic expression constructed using the independent support.

The probability of $v = i$ is the probability of being in a state where $f(v) = i$, where i is one of the possible values that v can be assigned and we do this. We tag every state in the model where $f(v) = i$ is satisfied. In doing so, we create a *reward* model for the RTL-DTMC. A reward is defined as a cost associated with being in various states of the DTMC. We assign a reward of 1 to the states we want to tag, and 0 otherwise. The *expected* value of the reward at any time step thus translates to the probability that a tagged state is reached in that step. Repeating this for all i gives the complete probability distribution of v . Similarly, we construct RTL-DTMC models for all statements and obtain probability distributions for all signals of interest.

We use our static analysis technique to obtain minimal independent supports for all output variables. This significantly reduces the size of the RTL-DTMCs leading to faster reward calculations, as compared to using subsets of I as independent supports for all output variables.

RTL-Delay-DTMCs express timing as a function of the state variables. The probability that the delay at an output signal meets a specified timing constraint is defined as *critical probability* and is the timing related invariant that we are interested in. We use the library function F obtained using delay macromodeling to annotate the RTL-Delay-DTMCs with delay-based rewards. All states where delay satisfies the timing constraint is *tagged* (reward=1). Therefore the expected reward represents the probability that timing constraint is met at any step. This reward model computes the required probabilistic invariant, *i.e.* critical probability.

Since the timing analysis is most meaningful for datapath, we obtain probabilistic delay distributions only for combinational variables. However, the probabilistic distributions of control signals (combinational and sequential) play a role in determining the delay of the datapath. Therefore we compute RTL-DTMC models for both datapath as well as control signals. We confine RTL-Delay-DTMC models to datapath signals.

D. Probabilistic Model Checking of the RTL-DTMCs

The next step in our flow is to compute the probabilistic invariant. We would like to find the probability of being in a tagged state, at the end of N transitions. This is equivalent to finding the expected value of the reward at the N^{th} time step. Simulation-based techniques typically explore a limited number of paths of length N , providing an incomplete analysis.

Probabilistic model checking is a formal verification technique for the analysis of stochastic systems. The expected value of a reward at the N^{th} time step is obtained by performing an exhaustive exploration of all the possible paths of length N . We use PRISM, a symbolic model checking tool that uses efficient algorithms and data structures based on

binary decision diagrams (BDDs). BDDs allow for compact representation and efficient manipulation of DTMCs, increasing the scalability of this tool considerably.

We start the DTMC system from a known initial state that we specify. For our experiments with the processor datapath, we use $N=3$ in our reward query for combinational variables. For sequential variables, the probabilities are empirically found to converge in about 10 steps, *i.e.* $N=10$.

E. Revising the design choice

The probabilistic timing invariants computed by SHARPE are provided as feedback to the RTL designer. These invariants are utilized to revise and if necessary, modify the RTL design. This process can be iterative until the timing constraints are satisfied. Additionally, the performance of different gate-level implementations of an RTL function can be explored by comparing the timing invariants obtained. Such information can be passed down to facilitate gate-level design.

SHARPE may be vulnerable to state space explosion that is associated with probabilistic model checking tools. In ongoing work, we are incorporating into SHARPE, several property-preserving reduction techniques [24] [25] as well as a compositional approach [26]. We believe that the scalability of SHARPE can be significantly improved through these approaches.

IV. EXPERIMENTAL RESULTS

We apply SHARPE to the datapath of the OR1200, an embedded RISC processor. We refer to this RTL design as RTL_1. We perform our experiments on a 3 GHz, 3.25 GB machine.

We first analyze the accuracy of the probability distributions computed through static analysis. For this, we consider both datapath and a few control signals. We consider a set of signals to serve as primary inputs whose probability distributions we define. We do not explicitly model the 32-bit instruction registers. All signals that are directly driven by the values stored in these registers are included in the set of primary inputs. We assume that primary input signals like *rst* and *mac_stall* take on their atypical values (equal to 1) once in 100 million clock cycles. Therefore, we assume that in each clock cycle, these signals are assigned a value of 1 with a probability of 10^{-8} . In the datapath, we consider the output signal *result* of a 6-bit ALU. The data signals *a* and *b*, that serve as synchronously arriving inputs to the ALU are modeled as primary inputs with uniform probability distribution.

TABLE I
ERROR OF COMPUTED PROBABILITY DISTRIBUTIONS.

Signals	Simulation Length		
	10^2	10^5	10^8
<i>if_freeze, ex_freeze</i> <i>lsu_unstall, flushpipe</i> <i>extend_flush, wb_freeze</i>	100%	100%	12.8%
<i>result</i>	43.2%	11.6%	4.5%

We use the RTL-DTMC models to determine that the control signals in Row 1 of Table I are assigned one of their infrequent values with a probability of 10^{-8} . *result* is found to be assigned all its 64 possible values with equal probability. Table I compares the percentage of error between the probability distributions estimated using SHARPE, against the values obtained through RTL simulations for different lengths.

As we increased the simulation length, the percentage error reduces significantly for both control and datapath signals, implying that our analytical estimation is of high-confidence. With simulation lengths less than 10^8 , we find the control signals are not assigned their infrequent value (rare event) even once and hence we record an error of 100%. The rare events can perhaps be neglected for purposes of statistical timing analysis. However, this establishes the accuracy of our static analysis approach.

For timing estimation, we obtain the delay macromodels for shifters (shift-rotators) and $2^K:1$ MUX as functions of the delay of basic 2:1 MUX blocks. For bitwise operators, we use the delay models of the corresponding basic blocks (AND, XOR or OR). For a 6-bit ripple carry adder (subtractor), we obtain the macromodel as described in Section III A 1). We fit a 5th order polynomial curve, shown in Figure 2, to the simulated delay measurements at the MSB, as a function of the *ripples*. The mean error between the delay predicted by this analytical model and the measured delay is less than 1% of the worst-case timing for the adder.

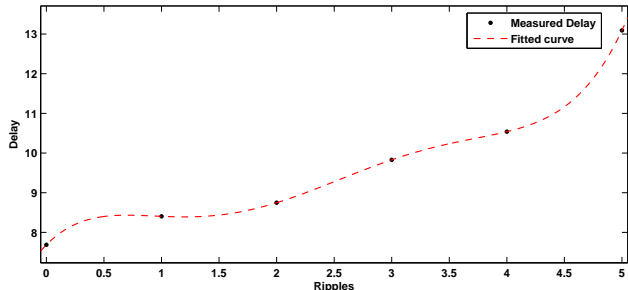


Fig. 2. Polynomial curve fit to adder delay measurements.

We estimate the timing invariants for the MSB of the ALU output signal, *result* using our RTL-Delay-DTMC model. We set the maximum timing delay observed through simulations as the worst-case time (WCT). For different timing constraints, we compare our results (Table II) with the critical probabilities (Section III C) obtained by gate-level simulations that utilize delay values from the NANGATE standard cell library. The estimates obtained through both approaches differ only by a maximum of 6.45%.

We now illustrate our technique for input signals with different arrival times as in Section 3.1.2. We modify the ALU module by introducing a 2:1 MUX after each of the inputs. This introduces independent timing offsets to their arrival times to the original ALU module (RTL_1) and we

TABLE II
CRITICAL PROBABILITIES OF *result* IN RTL_1.

Constraint (% of WCT)	Critical Probabilities		
	SHARPE	Gate-level simulations	Percentage Error(%)
100%	1.0	1.0	0
90%	0.9992	0.9929	0.63%
80%	0.9990	0.9748	2.50%
70%	0.9772	0.9180	6.45%
60%	0.9531	0.8993	5.98%

shall call this modified design, RTL_2. The calculated critical probabilities are listed in the Table III. Even in this case, the SHARPE estimates deviates from simulation-based estimates less than 6.05%.

TABLE III
CRITICAL PROBABILITIES OF *result* IN RTL_2.

Constraint (% of WCT)	Critical Probabilities		
	SHARPE	Gate-level simulations	Percentage Error(%)
100%	1.0	1.0	0
90%	0.9990	0.9810	1.83%
80%	0.9890	0.9562	3.43%
70%	0.9531	0.9122	4.48%
60%	0.9529	0.8985	6.07%

Let RTL_2P be a pipelined version of RTL_2, where registers are introduced after both the MUXes that offset the inputs. By definition, we consider only the delay from the register outputs triggered at the rising edge of clock. Therefore, the delay distribution curve of interest in RTL_2P is exactly the same as that of RTL_1 (Table II). This demonstrates how an RTL designer can make use of the probabilistic timing invariants to modify the RTL (introduce pipelining, for example) and obtain a better delay distribution curve.

We consider an alternate gate-level implementation of RTL_1, which has a carry lookahead adder, with 2 3-bit ripple carry adder blocks. We obtain the corresponding delay macromodel and use it to determine the probabilistic timing invariants. It is observed that only 93.2% of the inputs satisfy the 90% WCT constraint. This information prompts the gate-level designer to choose a ripple carry adder implementation.

We do not use post-synthesis gate-level netlists of the ALU module, since it would be optimized to meet a specific timing constraint. Instead, we aim to analyze the pre-synthesis critical probabilities which would aid in choosing a better than worst-case timing constraint.

Through static analysis, we have already established connectivity information between the various blocks. This is sufficient to define more complex delay macromodels that represent the effect of loading on delay, accurately. In our analysis, we do not account for the capacitive loading at the fanout of a block. We assume this for gate-level simulations as well to maintain a level platform for comparison of timing estimates.

In conclusion, we have presented SHARPE, a novel tool flow and methodology for generating probabilistic invariants at the higher levels of system design. In future work, we

plan to incorporate process variation, variations due to voltage scaling etc as a part of our tool flow. We will use state space reduction techniques and a compositional approach to improve the scalability of SHARPE.

REFERENCES

- [1] K. A. Bowman, M. Orshansky, and S. S. Sapatnekar, "Tutorial ii: Variability and its impact on design," in *Proc. of ISQED'06*, 2006, p. 5.
- [2] M. Berkelaar, "Statistical delay calculation," 1997.
- [3] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single pert-like traversal," in *Proc. of ICCAD'03*, 2003.
- [4] J.-J. Liou, K.-T. Cheng, S. Kundu, and A. Krstic, "Fast statistical timing analysis by probabilistic event propagation," in *Proc. of DAC'01*, 2001, pp. 661–666.
- [5] M. Orshansky and K. Keutzer, "A general probabilistic framework for worst case timing analysis," in *Proc. of DAC'02*, 2002, pp. 556–561.
- [6] N. Shanbhag, "Reliable and energy-efficient digital signal processing," in *Proc. of DAC'02*, 2002, pp. 830–835.
- [7] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Micro Conference*, Dec. 2003.
- [8] J. Jung and T. Kim, "Timing variation-aware high-level synthesis," in *Proc. of ICCAD'07*, 2007, pp. 424–428.
- [9] W.-L. Hung, X. Wu, and Y. Xie, "Guaranteeing performance yield in high-level synthesis," in *Proc. of ICCAD'06*, 2006, pp. 303–309.
- [10] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proc. of ISCA'08*, 2008.
- [11] L. de Alfaro, "How to specify and verify the long-run average behavior of probabilistic systems," in *LICS*, 1998, pp. 454–465.
- [12] T. S. Hoang, Z. Jin, K. Robinson, A. Mciver, and C. Morgan, "Probabilistic invariants for probabilistic machines," in *Proc. of ZB'03*, Springer, 2003, pp. 240–259.
- [13] S. Sambamurthy, J. Abraham, R. Tupuri, and S. Raghuram, "A robust top-down dynamic power estimation methodology for delay constrained register transfer level sequential circuits," in *Proc. of VLSID'08*, 2008, pp. 521–526.
- [14] T. Koyagi, M. Fukui, and R. Saleh, "Delay macromodeling and estimation for rtl," May 2008, pp. 2430–2433.
- [15] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, 2005.
- [16] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 2.0: A tool for probabilistic model checking," in *Proc. of QEST'04*, 2004, pp. 322–323.
- [17] A. Nandi and R. Marculescu, "System-level power/performance analysis for embedded systems design," in *Proc. of DAC'01*, 2001, pp. 599–604.
- [18] G. Norman, D. Parker, M. Kwiatkowska, S. K. Shukla, and R. K. Gupta, "Formal analysis and validation of continuous-time markov chain based system level power management strategies," in *Proc. of HLDVT'02*, 2002, p. 45.
- [19] K. Nepal, R. I. Bahar, J. L. Mundy, W. R. Patterson, and A. Zaslavsky, "Designing logic circuits for probabilistic computation in the presence of noise," in *Proc. of DAC'05*, 2005, pp. 485–490.
- [20] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge, "Opportunities and challenges for better than worst-case design," in *Proc. of ASP-DAC'05*, 2005, pp. 2–7.
- [21] N. Shanbhag, "Alternative computational models," March 2008.
- [22] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing of hardware description languages," in *Proc. of CHARME'99*, 1999, pp. 298–312.
- [23] A. Bogliolo, L. Benini, and G. De Micheli, "Regression-based rtl power modeling," *TODAES'00*, vol. 5, no. 3, pp. 337–372, 2000.
- [24] M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking," in *Proc. of CAV'06*, ser. LNCS, vol. 4114, Springer, 2006, pp. 234–248.
- [25] C. Baier, M. Grer, and F. Ciesinski, "Partial order reduction for probabilistic systems," in *Proc. of QEST'04*. IEEE Computer Society Press, 2004, pp. 230–239.
- [26] L. de Alfaro, T. Henzinger, and R. Jhala, "Compositional methods for probabilistic systems," in *Proceedings of the 12th International Conference on Concurrency Theory*, Jan. 2001, pp. 351–365.